Exploring Self-star Properties in Cognitive Sensor Networking

Pruet Boonma and Junichi Suzuki
Department of Computer Science
University of Massachusetts, Boston
{pruet, jxs}@cs.umb.edu

Abstract

Wireless sensor networks (WSNs) possess inherent tradeoffs among conflicting operational objectives such as data yield, data fidelity and power consumption. In order to address this challenge, this paper proposes a biologically-inspired framework to build cognitive WSN applications, which introspectively understand their conflicting objectives, find optimal tradeoffs with given constraints and autonomously adapt to dynamics of the network. The proposed framework, MONSOON, models an application as a decentralized group of software agents. This is analogous to a bee colony (application) consisting of bees (agents). Agents collect sensor data on individual nodes and carry the data to base stations. They perform this data collection functionality by autonomously sensing their local and surrounding network conditions and adaptively invoking biological behaviors such as pheromone emission, reproduction and migration. Each agent has its own behavior policy, as a gene, which defines how to invoke its behaviors. MONSOON allows agents to evolve their behavior policies via genetic operations such as crossover and mutation. Simulation results show that agents (WSN applications) exhibit the properties of self-configuration, self-optimization and self-healing and adapt to various dynamics of the network (e.g., node/link failures) by satisfying conflicting objectives under given constraints.

1. Introduction

Wireless sensor networks (WSNs) have inherent tradeoffs among conflicting operational objectives such as data yield, data fidelity and power consumption. For example, in data collection applications, hop-by-hop recovery is often applied for packet transmission in order to improve data yield (the quantity of collected data). However, this can degrade data fidelity (the quality of collected data; e.g., data freshness). For improving data fidelity, sensor nodes may transmit data to base stations with the shortest paths; however, data yield can degrade because of traffic congestion and packet losses on the paths.

In order to address this issue, the authors of the paper envision *cognitive* WSN applications that introspectively understand their conflicting objectives, find optimal tradeoffs under given constraints and autonomously adapt to dynamics of the network such as node/link failures. For making this vision a reality, this paper proposes a cognitive sensor networking framework, called MONSOON[1], which allows WSN applications to exhibit the following self-* properties:

- *Self-configuration*: allows WSN applications to automate their own configurations and self-organize into desirable structures and patterns (e.g., routing paths and duty cycles).
- *Self-optimization*: allows WSN applications to constantly seek improvement in their performance by adapting to changing network conditions with minimal human intervention.
- *Self-healing*: allows WSN applications to automatically detect and recover from disruptions in the network (e.g., node and link failures).

As an inspiration for the design strategy of MONSOON, the authors of the paper observe that various biological systems have developed the mechanisms necessary to realize the vision of MONSOON. For example, a bee colony self-organizes to satisfy conflicting objectives simultaneously for maintaining its well-being [8]. Those objectives include maximizing the amount of collected honey, maintaining the temperature in a nest and minimizing the number of dead drones. If bees focus only on foraging, they fail to ventilate their nest and remove dead drones. Given this observation, MONSOON applies key biological mechanisms to implement cognitive WSN applications.

Figure 1 shows the architecture of MONSOON. The MONSOON runtime operates atop TinyOS on each node. It consists of two types of software components: *agents* and *middleware platforms*, which are modeled after bees and flowers, respectively. Each application is designed as a decentralized group of agents. This is analogous to a bee colony (application) consisting of bees (agents). Agents collect sensor data on platforms (flowers) atop individual nodes, and carry the data to base stations on a hop-by-hop basis, in turn, to a backend server (the MONSOON server in Figure 1), which is modeled after a nest of bees.

Agents perform this data collection functionality by autonomously sensing their local and surrounding network conditions and adaptively invoking biological behaviors such as pheromone emission, replication, reproduction, migration and death. A middleware platform runs on each node, and hosts one or more agents (Figure 1). It provides a series of runtime services that agents use to perform their functionalities and behaviors.

MONSOON implements a constraint-based evolutionary adaptation mechanism for agents. Each agent has its own

[1] Multiobjective Optimization for a Network of Sensors using an evOlutionary algOrithm with coNstraints

behavior policy, as a *gene*, which defines when to and how to invoke its behaviors. MONSOON allows agents to evolve their behavior policies via genetic operations (mutation and crossover) and simultaneously adapt them to conflicting objectives with associated constraints. A constraint is defined as an upper or lower bound for an objective. For example, a tolerable (lower) bound may be defined for data fidelity. Currently, MONSOON considers six objectives related to data yield, data fidelity and power consumption.

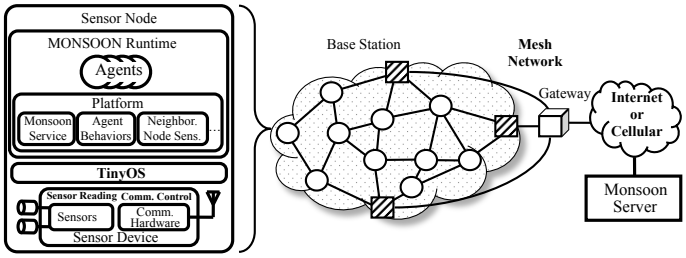

Figure 1. The Architecture of MONSOON

2. The MONSOON Runtime

MONSOON is currently designed to implement data collection applications. An agent is initially deployed with a randomly-generated behavior policy on each node. Each agent collects sensor data on a node periodically (i.e., at each duty cycle) and carry the data toward a base station.

2.1. Agent Behaviors

Each agent implements seven behaviors and performs them in the following sequence at each duty cycle.

Step 1: Energy gain. Each agent collects sensor data and gain *energy*. In MONSOON, the concept of energy does not represent the amount of physical battery in a node. It is a logical concept that impacts agent behaviors. Each agent updates its energy level with a constant energy intake (E_F):

$$E(t) = E(t-1) + E_F \tag{1}$$

$E(t)$ and $E(t-1)$ denote the energy levels in the current and previous duty cycles.

Step 2: Energy expenditure and death. Each agent consumes a constant amount of energy to use computing/networking resources available on a node (e.g., CPU and radio transmitter). It also expends energy to invoke its behaviors. The energy costs to invoke behaviors are constant for all agents. An agent dies due to energy starvation when it cannot balance its energy gain and expenditure. The death behavior is intended to eliminate the agents that have ineffective behavior policies. For example, an agent would die before arriving at a base station if it follows a too long migration path. When an agent dies, the local platform removes the agent and releases all resources allocated to it[2].

[2]If all agents are dying on a node at the same time, a randomly selected agent will survive. At least one agent runs on each node.

Step 3: Replication. Each agent makes a copy of itself in each duty cycle. A replicated (child) agent is placed on the node that its parent resides on, and it inherits the parent's behavior policy (gene). A replicating (parent) agent splits its energy units to halves ($\frac{E(t)-E_R}{2}$), gives a half to its child agent, and keeps the other half. E_R denotes the energy cost for an agent to perform the replication behavior. A child agent contains the sensor data that its parent collected, and carries it to a base station on a hop by hop basis.

Step 4: Swarming. Agents may swarm (or merge) with others at intermediate nodes on their ways to base stations. On each intermediate node, each agent decides whether it migrates to a next-hop node or waits for other agents to arrive at the current node and swarm with them. This decision is made based on the migration probability (p_m). If an agent meets other agents during a waiting period, it merges with them and contains the sensor data they carry. It also uses the behavioral policy of the best one in the aggregating agents in terms of operational objectives. (See Section 3. on how to find the "best" agent.) The swarming behavior is intended to save power consumption by reducing the number of data transmissions. If the size of data an agent carries exceeds the maximum size of a packet, the agent does not consider the swarming behavior.

Step 5: Pheromone sensing and migration. On each intermediate node toward a base station, each agent chooses a migration destination node (next-hop node) by sensing three types of *pheromones* available on the local node: base station, migration and alert pheromones.

Each base station periodically propagates a base station pheromone to individual nodes in the network. Their concentration decays on a hop-by-hop basis. Using base station pheromones, agents can sense where base stations exist approximately, and move toward them by climbing a concentration gradient of base station pheromones.

Agents emit migration pheromones on their local nodes when they migrate to neighboring nodes. Each migration pheromone references the destination node an agent has migrated to. Agents also emit alert pheromones when they fail migrations within a timeout period. Migration failures can occur because of node failures due to depleted battery and physical damages as well as link failures due to interference and congestion. Each alert pheromone references the node that an agent could not migrate to. Each of migration and alert pheromones has its own concentration. The concentration decays by half at each duty cycle. A pheromone disapears when its concentration becomes zero.

Each agent examines Equation 2 to determine which next-hop node it migrates to.

$$WS_j = \sum_{t=1}^{3} w_t \frac{P_{t,j} - P_{t_{min}}}{P_{t_{max}} - P_{t_{min}}} \tag{2}$$

An agent calculates this weighted sum (WS_j) for each